NPS-CS-07-005

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Experiments with Sun Java Real-Time System - Part II

by

M. Auguston, T. S. Cook, D. Drusinsky, J. B. Michael
T. W. Otani, and M. Shing

11 May 2007

Approved for public release; distribution is unlimited

Prepared for: Missile Defense Agency
7100 Defense Pentagon
Washington, D.C. 20301-7100

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Provost

This report was prepared for the Missile Defense Agency and funded by the Missile Defense Agency.

This report was prepared by:

_____
Thomas Otani
Associate Professor of Computer Science
Naval Postgraduate School

Reviewed by:

_____
Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____
Dan C. Boger
Interim Associate Provost and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** May 2007	**3. REPORT TYPE AND DATES COVERED** Technical Report	
4. TITLE AND SUBTITLE: Title (Mix case letters) Experiments with the Sun Java Real-Time System – Part II			**5. FUNDING NUMBERS**
6. AUTHOR(S) Mikhail Auguston, Thomas S. Cook, Doron Drusinsky, James Bret Michael, Thomas W. Otani, and Man-Tak Shing			MD7080101P0630
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-07-005
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Missile Defense Agency, 7100 Defense Pentagon, Washington, DC 20301-7100			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This is our second report on the Sun Java Real-Time System (RTS) experiments. In this report we describe the findings on RTS 2.0 beta release that includes the real-time garbage collector (RTGC). We performed a number of experiments to determine whether the availability of RTGC will result in a better software architecture for the Global Integrated Fire Control System (GIFC)—a component of the C2BMC element of the Ballistic Missile Defense System (BMDS).

Our experiment shows that it is possible to use only the Real-Time Java threads that use the heap memory for the GIFC software. SUN RTJ 2.0 gives programmers more control over the priority of the garbage collection. We developed a real-time monitor design pattern to support the implementation of time-constrained computations that use the heap memory, and a methodology to determine the RTS run-time parameters (thread priorities, memory usage, process load, and task deadlines) necessary for the timely execution of these time-constrained computations.

14. SUBJECT TERMS Real-time system, Java programming language, Garbage collection, Ballistic Missile Defense System, Global Integrated Fire Control, Advanced Battle Manager			**15. NUMBER OF PAGES** 24 **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Experiments with Sun Java Real-Time System —Part II

M. Auguston, T. S. Cook, D. Drusinsky, J. B. Michael, T. W. Otani, and M. Shing

Abstract

This is our second report on the Sun Java RTS. In this report we describe the findings on RTS 2.0 beta release that includes the real-time garbage collector (RTGC). We performed a number of experiments to determine whether the availability of RTGC will result in a better software architecture for the Global Integrated Fire Control System (GIFC)—a component of the C2BMC element of the Ballistic Missile Defense System (BMDS).

Our experiment shows that it is possible to use only the Real-Time Java threads that use the heap memory for the GIFC software. SUN RTJ 2.0 gives programmers more control over the priority of the garbage collection. We developed a real-time monitor design pattern to support the implementation of time-constrained computations that use the heap memory, and a methodology to determine the RTS run-time parameters (thread priorities, memory usage, process load, and task deadlines) necessary for the timely execution of these time-constrained computations.

1.0 Overview

In our first report [COOK], we concluded that it is preferable to use only the Real-Time Java threads that use the heap memory and not the no-heap real-time threads (NHRTT) for the GIFC software, due to the difficulties in writing correct Java programs using the no-heap real-time threads. However, such architecture cannot be implemented by using RTS 1.0. Further experiments are needed to determine if the preferred architecture can be implemented with the RTS 2.0, which will give programmers more control over the priority of the garbage collection. RTS 2.0 alpha version was released in August, 2006 and the beta version in December, 2006. RTGC is supported in the new version, and we performed experiments to explore viable software architectures for the GIFC software.

Although the alpha release RTS 2.0 has made a number of improvements over RTS 1.0, the RTGC falls short of our expectations and needs. The beta version (RTS 2.0 b31) was released in December, 2006. Our experiments indicated that critical improvements made in the beta version did meet our software requirements. As the alpha version is now obsolete, we will not describe the experiments we performed with the alpha release in this report.

2.0 Software and Hardware Configuration

In order to run the RTS 2.0 beta version, we are required to update our Solaris machine. The hardware is SunBlade 2500. The operating system is Solaris 10 (11/06) and the RTS 2.0 version is b31 (12/06).

3.0 RTS v2.0 Real-Time Garbage Collector

The real-time garbage collector (RTGC) provided in Sun Java RTS 2.0 has made critical improvements over the previous versions. It is now fully concurrent, and it can be preempted by the application's real-time threads with a higher priority. Unlike the previous versions, there is no stop-the-world phase where the execution of the garbage collection suspends application threads unless the memory is completely depleted. Even in such situation, the effect to the critical threads (those with a priority higher than the one for the RTGC) is minimal. By properly tuning the values for two runtime parameters—RTGCCriticalPriority and RTGCCriticalReservedBytes—we will be able to achieve the desired balance of deterministic behavior of the critical threads and an overall throughput of the application.

The new RTGC has the characterictic of dynamically changing its priority to achieve the right balance between the determinism of critical threads and the overall throughput. With the alpha release, we set a fixed priority to the RTGC and assign higher priorities to critical threads. This set up however did not prevent the RTGC to block critical threads. With the newer beta release, we specify the threshold instead of setting a specific priority to the RTGC. Any thread with a priority above this threshold is considered critical. The RTGC starts with the lowest priority for the real-time threads.

In the following subsections, we describe the effect of the two runtime parameters in detail. These two parameters are new to the beta release.

3.1 RTGCCriticalPriority

The RTGCCriticalPriority runtime parameter is most significant in the RTS V2 release for ensuring the determinism of time-critical threads. A thread with the assigned priority higher than RTGCCriticalPriority is called the *critical real-time thread*. RTGC starts running at RTGCNormalPriority (whose default value is the minimum priority for the real-time threads). The auto-tuning mechanism attempts to start RTGC soon enough so

that the garbage collection completes before reaching the memory threshold (RTGCCriticalReservedBytes), which will result in bumping up the priority of RTGC to RTGCCriticalPriority.

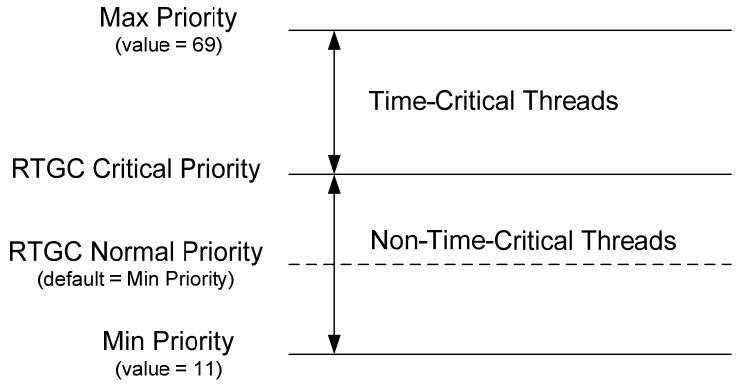

Figure 1. Java real-time thread classification

3.2 RTGCCriticalReservedBytes

To aid the RTGC in ensuring the deterministic behavior of all the time-critical threads, the programmer needs to specify the second runtime parameter RTGCCriticalReservedBytes (the default value is 0). When the free memory becomes less than the value set for RTBCCriticalReservedBytes, RTGC runs at RTGCCriticalPriority, using all CPU cycles not used by time-critical threads. This prevents all other threads (non-time-critical real-time threads and non-real-time threads) from allocating CPU cycles and memory, and caused them to be blocked. It is important to be aware that critical threads with a higher priority can still get blocked by the lower priority RTGC if there is not enough memory for the critical threads to run. In general, we want to set the RTGCCriticalReservedBytes just high enough to ensure that the critical threads do not get preempted by the RTGC due to lack of free memory. If RTGCCriticalReservedBytes is set too high, the RTGC will run more frequently, thereby preventing the lower priority threads from running. This will reduce the overall throughput. The important points to remember regarding the value for RTGCCriticalReservedBytes are as follows:

- `RTGCCriticalReservedBytes too high --> lower throughput`
- `RTGCCriticalReservedBytes too low --> determinism compromised`

4.0 Experiments

In this section, we present the main experiments we performed. As promised in our first technical report [COOK], we have repeated the experiments described there. In addition, we performed new experiments to further explore different software architectures that are suitable for the new RTGC.

All of the experiments described in this report were run under RTJ 2.0 beta release. We set the value for RTGCCriticalReservedBytes to 0 for all the experiments we performed.

We set the value to 0 so we can investigate the ideal configuration for the GIFC that achieves the maximum throughput. The new RTGC performed in a satisfactory manner and has met our expectations.

4.1 Experiments No 1 to No 4

These are the experiments we performed and reported in our previous technical report [COOK]. We ran these experiments again using Java RTS 2.0 to verify that we do not encounter any peculiar behavior under the new RTS version. We did not observe any anomaly when we ran these previous experiments using Java RTS 2.0.

In Experiment No. 1, a real-time thread, with the highest possible priority, creates a linked list of nodes. We confirmed that this thread does get preempted even if its priority is higher than the one for RTGC when the available memory is exhausted. Experiment No. 5 is an updated version of Experiment No. 1, which we describe in the next section. Experiments No. 2 and No. 3 used NoHeapRealtimeThread objects, and as such, they are somewhat moot under RTS 2.0 since our recommended software architecture does not use no-heap real-time threads.

In Experiment No. 4, we used only regular real-time threads (i.e., instances of the RealtimeThread class) dividing them into nominal and stateless discriminators. If a stateless discriminator can finish its task within the designated deadline, the actual result is reported. If it cannot finish its task within the deadline, the nominal result is reported. We observed the expected behavior of getting more actual results when we increase the values for the deadlines. Experiment No. 7 we describe in this report is an improved version of Experiment No. 4. For more details on Experiments No. 1 to No. 4, please refer to our previous technical report [COOK].

4.2 Experiment No. 5: Interaction between RTGC and RealtimeThreads

The purpose of this experiment is to check the thread priority relationship between the real-time threads (instances of RealTimeThread and its subclasses) and the real-time garbage collector (RTGC). We define a descendant of RealtimeThread named RTGC_Tester. A RTGC_Tester simulates real-time computation by creating a linked list of N nodes with each node having an array of 500 BigInteger objects, as shown in the following pseudo-code:

```
List<Node> myList = new LinkedList<Node>();
Node node;

for (int i = 0; i < N; i++) {
       node = new Node(i);
       myList.add(node);
}
```

By varying the values for N, we can study the impact of the real-time garbage collector to the running program when the garbage collection kicks in.

The main driver creates and runs a number (M) of RTGC_Tester objects, where the value for M is an input to the program. RTGC_Tester objects are executed in sequence, and we track the elapsed time of each object in completing its execution. The following is the pseudo-code:

```
for (int i = 0; i < M; i++) {

       myThread = new RTGC_Tester(...);

       myThread.start();
       myThread.waitTillCompletion():

       elapsedTime[i] = myThread.getElapsedTime();
}

for (int i = 0; i < M; i++) {
       System.out.println("Elapsed Time " + elapsedTime[i]);
}
```

As more and more RTGC_Tester objects are executed, memory is consumed, and depending of the values of M and N, garbage collection would take place. We want to see how the real-time garbage collection would affect the execution of RTGC_Tester objects. We ran the test driver in two ways. In the first way, we set the priority of RTGC_Tester objects to the highest value as follows:

```
myThread =
    new RTGC_Tester(
       new PriorityParemeters(PriorityScheduler.instance(),
                                                          getMaxPriority()),
       null);
```

And in the second way, we set their priority to the lowest possible value:

```
myThread =
    new RTGC_Tester(
       new PriorityParemeters(PriorityScheduler.instance(),
                                                          getMinPriority()),
       null);
```

We set the RTGCNormalPriority to 40 and the RTGCCriticalReservedBytes to 0. In doing so, the RTGC will not block the RTGC_Tester objects with the highest priority as long as there is free memory in the heap, but will preempt the RTGC_Tester objects with the lowest priority periodically to reclaim the memory in the heap.

It is important to notice that there is exactly one RTGC_Tester object running at any single instance of time. Whether this object has a priority higher than the one for the real-time garbage collector becomes irrelevant when the free memory is completedly exhausted. When there is no more free memory, the garbage collector will preempt any real-time thread (regardless of their priority) to reclaim memory. We observed this behavior is our test runs. The priority assigned to the RTGC_Tester objects is irrelevant when the free memory is exhausted. Notice the two tables display the similar results.

Table 1. Experiment 5 Test Results with Max priority RTGC_Tester

M (repeat count)	No of times GC occurred	No of spikes in elapsed time	Elapsed time of the interrupted RTGC_Tester	Comment
100	0	0	--	Elapsed time for the uninterrupeted RTGC_Tester is approximately 3 ms.
200	1	1	143 ms	
300	1	1	144 ms	
500	3	3	151 ms 141 ms 141 ms	
1000	6	6	174 ms 142 ms 138 ms 139 ms 137 ms 146 ms	

Table 2. Experiment 5 Test Results with Min priority RTGC_Tester

M (repeat count)	No of times GC occurred	No of spikes in elapsed time	Elapsed time of the interrupted RTGC_Tester	Comment
100	0	0	--	Elapsed time for the uninterrupeted RTGC_Tester is approximately 3 ms.
200	1	1	144 ms	
300	1	1	144 ms	
500	3	3	144 ms 144 ms 140 ms	
1000	6	6	145 ms 141 ms 143 ms 142 ms 140 ms 140 ms	

4.3 Experiment No. 6: Mixture of Critical and Non-Criticial Realtime Threads

Experiment No. 5 is an extreme case where the program includes one type (either all critical or all non-critical) of running threads. The sole purpose of Experiment No. 5 was to verify the behavior of the RTGC interrupting any thread when the memory is exhausted.

In this experiment, we test a more realistic configuration where a mixture of critical and non-critical threads coexist in a running program. This configuration is closer to our proposed architecture of using high-priority Nominal and low-priority Stateless discriminators. For this experiment, we define two real-time thread classes: RTGC_Nominal and RTGC_Stateless. An instance of the RTGC_Nominal class simulates a Nominal object by spending time doing computation without allocating any memory in heap (uses only a local variable). An instance of the RTGC_Stateless class simulates a Stateless object by allocating a linked list of 2000 nodes with each node holding 500 BigInteger objects.

The main class of the experiment will create N RTGC_Nominal and N RTGC_Stateless threads in the initialization phase (where N is an input to the program) and then run the 2N threads concurrently. The RTGC_Nominal threads are run in the highest priority and the RTGC_Stateless in the lowest real-time thread priority. The priority of the RTGC is set to 40 as a runtime option. For each thread, we track its elapsed time.

Because the RTGC_Stateless threads allocate heap memory, we expect them to be interrupted and paused to wait for the garbage collector to complete its work, while the RTGC_Nominal threads sees no interruptions. The test runs confirmed our expectation. The following table shows some of the test results.

Table 3. Experiment 6 Test Results

N (repeat count)	No of times GC occurred	Minimum and Maximum elapsed times of RTGC_Nominal threads		Minimum and Maximum elapsed times of RTGC_Stateless Threads	
100	7	0 ms, 202417 ns	0 ms, 311083 ns	15 ms, 589583 ns	693 ms, 565833 ns
200	17	0 ms, 202333 ns	0 ms, 332916 ns	16 ms, 131334 ns	830 ms, 300750 ns
300	25	0 ms, 202750 ns	0 ms, 327750 ns	15 ms, 481501 ns	886 ms, 804750 ns
500	43	0 ms, 202500 ns	0 ms, 330583 ns	15 ms, 264834 ns	937 ms, 5332 ns
1000	90	0 ms, 201000 ns	0 ms, 455500 ns	15 ms, 188917 ns	882 ms, 228249 ns

4.4 Experiment No 7: Modified Experiment No 4

As stated in the first report [COOK], working with NHRTT is not easy. There are many pitfalls and hurdles software engineers and programmers must jump. Our goal with the new RTGC is to implement all-heap real-time thread architecture. In this All-Heap Design, instead of running the Discriminator as NHRTT, we will run it as a regular RTT, but assign a scheduling priority higher than RTGCCriticalPriority.

In Experiment No 4, we used a deadline miss handler to process the timeout situation in which the DiscriminatorStateless is not able to compute the result within the allocated time period (Figure 2). Because the deadline miss handler is associated to the DiscriminatorStateless that has a priority lower than the one for the RTGC, there is a possibillity of RTGC interrupting/delaying the asyncronous transfer of control to the deadline miss handler.

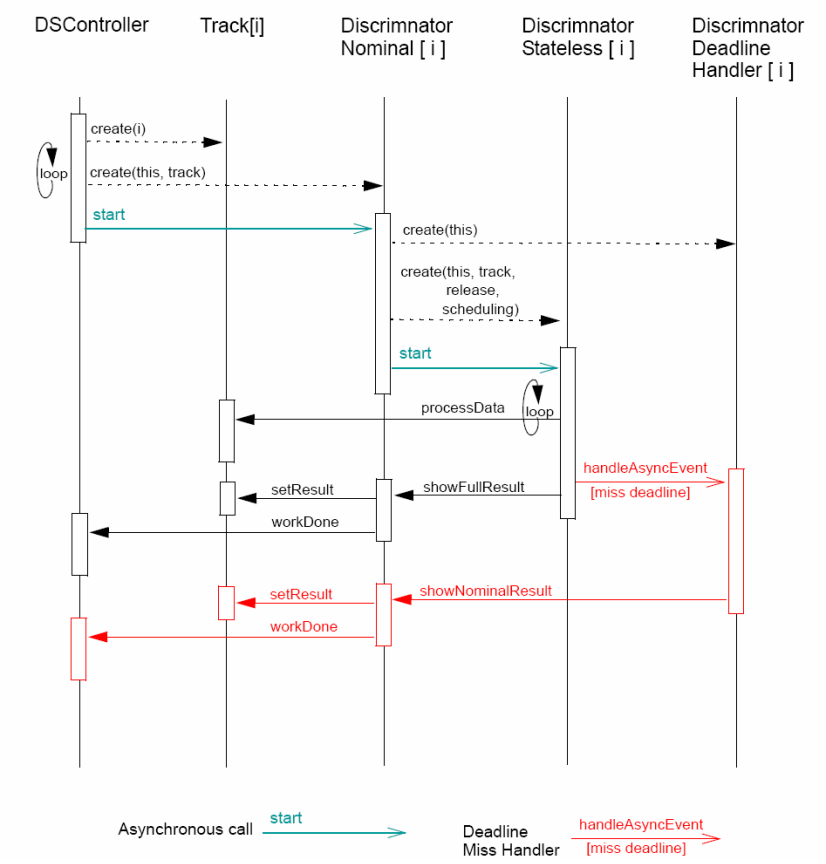

Figure 2. Sequence diagram of the original design

To avoid this undesirable possibility, one member of the Sun Java RTS project team suggested an alternative architecture to associate the deadline miss handler to a higher priority DiscriminatorNominal. To do this, we need to use a timer, specifically, a OneShotTimer. We designate the deadline to this OneShotTimer object by specifying a RelativeTime such as 20ms. When the set time is up, the OneShotTimer will trigger an event that allows the deadline handler to process the missed deadline.

4.4.1 Software Architecture

We divide real-time threads into the two groups: those with a priority higher and those lower than the one for the RTGC. We call them the critical and noncritical threads, respectively. The key aspect of this architecture is that only the noncritical threads allocate memory in the heap. In our particular case, only the Stateless instances will allocate the heap memory. This architecture ensures that the critical threads will not get preempted by the RTGC, thus guaranteeing the determinism of the critical threads.

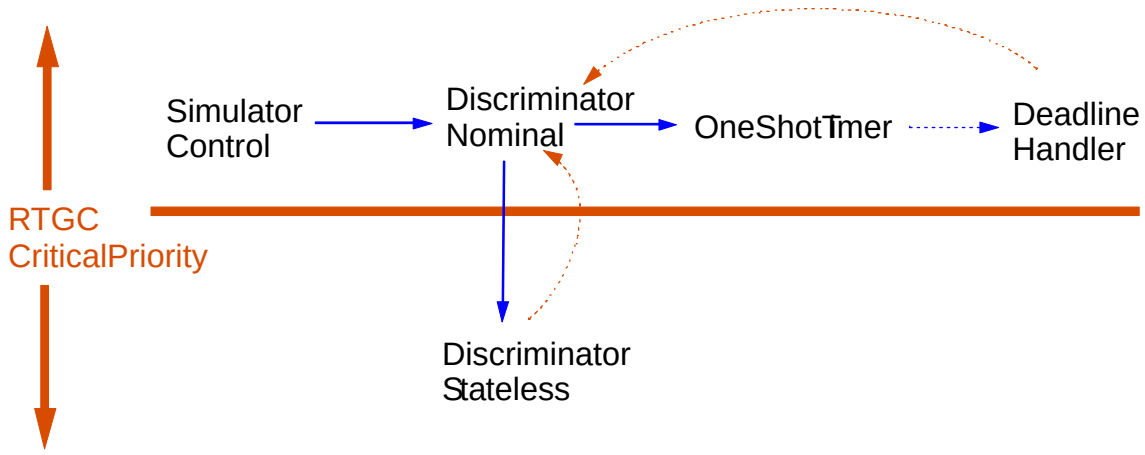

Figure 3. Collaboration diagram of the revised design

4.4.2 SimulatorControl

The main controller of the program creates N tracks, and for each track created, an instance of DiscriminatorNominal is assigned to it for discrimination. Each DiscriminatorNominal instantiates an instance of low priority DiscriminatorStateless to do the actual work of discrimination using the heap memory, and a high priority OneShotTimer to monitor the time taken by the DiscriminatorStateless thread.

SimulatorControl is a RealtimeThread and its run method is defined as follows:

```
public void run( ) {

       for (int i = 0; i < N; i++) {
              Track node = new Track(i);

              DiscriminatorNominal disc
                     = new DiscriminatorNominal(this, node);

              nominal[i] = disc;

              dicriminatorCnt++;
       }

       for (int i = 0; i < N) {
              nominal[i].start();

              /* A */
              /* Place delay here */
       }
}
```

We are using an array to keep track of nominal discriminators. Every index position of this array is a non-null value as it points to an instance of the DiscriminatorNominal class. When the DiscriminatorNominal finishes its computation, it calls the Simulator-Control's method (named workDone) to report the completion of discriminaton. This results in setting the corresponding index position to null, thereby turning the used heap memory into garbage.

At Point A in the code, after a nominal discriminator is started, we can place a time delay. Placing no delay means the program will run all nominal discriminators simultaneously. This could lead to an OutOfMemory exception when N becomes larger than a certain threshold. The reason is because the priority of SimulatorControl is higher than the one for RTGC. As it creates and starts more and more nominal discriminators, more and more memory gets consumed but there is no garbage to collect because there is no index position in the array that is set to null. In other words, from our experiment, the nominal discriminators never have a chance to call the workDone( ) method.

If we place certain amount of delay at Point A in the code, then it becomes possible for the nominal discriminators to call the SimulatorControl's method (named workDone) to turn themselves and memory allocated by the corespondingstateless discriminators into garbage for the RTGC to collect.

4.4.3 DiscriminatorNominal

A DiscriminatorNominal object performs the discrimination operation on a given track. The actual work of discrimination is done by DiscriminatorStateless. The deadline is set by designating the time duration (RelativeTime that specifies the time duration such as 2 ms) using a OneShotTimer. When the time is up, its associated asynchronous event handler DiscriminatorDeadlineHandler is used to report back to DiscriminatorNominal.

DiscriminatorNominal can get the result in two ways. The first is the full result, that is, the actual computation result received from DiscriminatorStateless. In this case, the OneShotTimer object is killed. The second is the nominal result. This result is used when the timeout occurs. In this case, the associated DiscriminatorStateless is killed.

4.4.4 DiscriminatorStateless

An instance of this class does the actual work of discrimination by interacting with its associated Track object. In this program, we simulate the discrimination activity by calling a method of the Track object. This stub method will go through a "dummy" computation loop and consume 40,000 bytes of heap memory. When the computation is complete, it calls its controlling DiscriminatorNominal to report the result.

4.4.5 DiscriminatorTimeoutHandler

When the time duration set for the OneShotTimer is up, it calls its controlling DiscriminatorNominal to report that the nominal result must be used.

4.4.6 Test Results

We ran tests by varying the number of discriminators, deadlines, and the pause time between the creation of discriminators (Point A in the code). The first set of tests is run by placing no delays (delay time = 0). The test results are shown in Table 4.

Table 4: No delay between the creation of discriminators

Deadline (ms)	N (# of discriminators)	Result (# of timeouts)
20	100	79 ~ 100
	200	200
	500	500
	1000	1000
	1500	OutOfMemory
50	100	28 ~ 96
	200	142 ~ 200
	500	500
	1000	1000
	1500	OutOfMemory
100	100	0 ~ 60
	200	35 ~ 200
	500	500
	1000	1000
	1500	OutOfMemory
500	100	0

	200	0
	500	184 ~ 434
	1000	998 ~ 1000
	1500	OutOfMemory

As expected, the table shows that as we increase the deadline, the number of timeouts decreases. When we increase the deadline the discriminators have more time to complete its computation. This will result in having a few number of timeouts for the same number of discriminators. For example, with 200 discriminators, we see anywhere from 35 to 200 occurences of timeouts when the deadline is set to 100 ms. When the deadline is increased to 500 ms, then we we see no timeouts at all.

In the second set of tests, we place a delay at Point A in the code. By placing a delay, we will be able run a larger number of discriminators without getting an out of memory exception because the RTGC will be able to reclaim garbage. With no delay, the run method of SimulatorControl never gets interrupted, and there will be no null pointers in the nominal array. With a delay, the run method can get interrupted and the nominal discriminators get a chance to call the workDone method of SimulatorControl. The workDone method will reset the content of the nominal array, at the index position that corresponds to the calling nominal discriminator, to null. This will result in the RTGC reclaiming memory allocated by the corresponding stateless discriminator. As heap memory spaces are recycled, we can avoid the OutOfMemory exceptions we've seen in the first set of tests.

Table 5: Delay of 5 ms between the creation of discriminators

Deadline (ms)	N (# of discriminators)	Result (# of timeouts)
20	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory
50	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory
500	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory

Table 5 shows the results of test runs when the delay time is set to 5 ms . When the number of discriminators (N) is 1500, we still get an OutOfMemory exception regardless the values for the deadline. This means the 5 ms delay time is simply not long enough for the run method to be interrupted and the nominal discriminator to get a chance to call the workDone method. If we set the deadline to a larger number, such as 500 ms, the RTGC does preempt SimulatorControl, but since the workDone method is never executed, there's no garbage the RTGC can collect.

Table 6 shows the results of test runs when the delay time is set to 50 ms, which gives enough time the the RTGC to reclaim the heap memory in between successive runs of the DiscriminatorNominal threads

Table 6: Delay of 50 ms between the creation of discriminators

Deadline (ms)	N (# of discriminators)	Result (# of timeouts)
20	100	0
	200	0
	500	0
	1000	0
	1500	1 ~ 5 (1 GC)
50	100	0
	200	0
	500	0
	1000	0
	1500	0 (1 GC)
500	100	0
	200	0
	500	0
	1000	0
	1500	0 (1 GC)

5.0 Conclusion and Recommendation

With the most recent Java RTS 2.0, we can assign a priority to the RTGC as a runtime option. A real-time thread with a priority higher than the RTGC priority will not get preempted by the RTGC unless the heap memory is completely exhausted. This architecture allows the programmers to divide the workload into critical and noncritical threads, with the former having a higher and the latter a lower priority than the RTGC priority.

We repeated Experiments No. 1 to No. 4 that are reported in our previous Technical Report with the Java RTS 2.0 to verify the new system is backward compatible. We performed Experiments No. 5 and No. 6 to confirm our understanding of how the new system works is correct. Finally, we carried out Experiment No. 7 to test the feasibility of our proposed design pattern for the GIFC.

The results of Experiment No. 7 confirms the viability of our proposed design pattern. The design pattern calls for two types of real-time threads—nominal and stateless—to carry out mission-critical tasks. The nominal threads run at a priority higher and the stateless threads lower than the one for the RTGC, respectively. Only the stateless threads allocate heap memory. The key aspect in our design pattern is the use of timeout handler. The nominal delegate the actual computation task to the stateless. If the stateless completes the designated task before the set deadline, it returns the result to the nominal. If the deadline is past without the stateless completing the task, the timeout handler notifies the nominal. In this case, the nominal will perform a table lookup for a value to be used in lieu of the real result.

The goal, when implementing our proposed design for the actual program, is to set the necessary parameters so the number of timeouts (T) is minimized. In an ideal situation, it is 0, that is, no timeouts occur. We observe three parameters are important, as demonstrated in Experiment No. 5. They are the pause time (P) between the creations of nominals, the deadline (D) for the stateless to complete the designated task, and the memory usage (M) of the stateless.

If we set P = 0, then the value of M determines the total number of nominals (N) and their corresponding stateless and timeout handlers that can be executed concurrently without causing an OutOfMemory exception. (In Experiment No. 7, we used M = 40,000 bytes.) By increasing the value for D, we can decrease the number of timeouts to reach the point where T would be 0. Table 1, for example, shows that by setting D = 500, we can run 200 discriminators without any occurence of timeouts.

In a typical real-time application, the upperbound for the value of D is given as a system requirement. That is, we want a guaranteed performance of completing a critical task in no more than D time units. If D is given, we can attain T = 0 by determining the appropriate values for P and M. If the upperbound for M is known, then we can increase the value of P until T becomes 0. If the upperbound for P is known, then we can determine the maximum value for M while maintaining T = 0.

To apply our proposed design pattern for the GIFC, we need an upperbound on the memory requirements for the task of discriminating and correlating a track. This gives us the value for M. The interarrival time of objects to track gives us the value for P. From these two values, we can guarantee no timeouts, T = 0, by determining suitable values for D and N. The values for D and N are proportional. If the deadline is longer, then we can ran a larger number of discriminators. If we want to run a larger number of discriminators, then we need to increase the deadline to maintain T = 0.

Table 7 summarizes the concepts. The "You Can/Need To" column specifies what the system designer needs to or can do in order to achieve no timeouts (T = 0) for the given parameters listed in the Given column.

Table 7: Methods to determine 100% completion of the stateless computation

To Achieve T = 0	
Given the values for	**You Can/Need To**
P and D	Determine the maximum value for M. If a discriminator requires less than this value, then we can achieve T = 0. If a dis criminator requires more, then we need to adjust the values of P or D to get T = 0.
P and M	Determine the threshold value for D. If this value is not acceptable, then we need to increase the value for P.
D and M	Determine the threshold value for P. Any value below this threshold will increase the occurences of timeouts.
T - timeouts; P - pause time (interarrival time); M - memory requirements per discriminator; D - deadline	

6.0 References

[BOLL] Bollella, Greg., et. al., *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[COOK] Cook, Tom., et. al., *Design of Preliminary Eperiments with the Sun Java Reat-Time System*, Naval Postgraduate School, Dept of Computer Science Technical Report, NPS-CS-06-010, May, 2006.

[DIBB] Dibble, Peter C., *The Real-Time Java Platform Programming*, Prentice-Hall, 2002.

[WELL] Wells, Andy, *Concurrent and Real-Time Programming in Java*, John Wiley & Sons, 2004.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5100

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Butch Caffall
   NASA IV&V Facility
   Fairmont, WV 26554

5. Mr. Richard Ritter
   Missile Defense Agency
   Washington, DC 20301-7100

6. Mr. John Shottes
   Missile Defense Agency
   Washington, DC 20301-7100

7. LTC Jason Stine
   Missile Defense Agency
   Washington, DC 20301-7100

8. LTC Thomas Cook
   Naval Postgraduate School
   Monterey, CA 93943

9. Dr. Doron Drusinsky
   Naval Postgraduate School
   Monterey, CA 93943

10. Dr. Bret Michael
    Naval Postgraduate School
    Monterey, CA 93943

11. Dr. Thomas Otani
Naval Postgraduate School
Monterey, CA 93943

12. Dr. Man-Tak Shing
Naval Postgraduate School
Monterey, CA 93943

13. Mr. Scott Pringle
Missile Defense National Team
Crystal City, VA

14. Mr. Erik Stein
Missile Defense National Team
Crystal City, VA

15. Mr. Tim Trapp
Missile Defense National Team
Crystal City, VA

16. Ms. Deborah Stiltner
Missile Defense National Team
Crystal City, VA

17. Mr. Steve Raque
NASA IV&V Facility
Fairmont, WV 26554

18. Mr. Marcus Fisher
NASA IV&V Facility
Fairmont, WV 26554